The Farm

Welcome to The Farm on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

The error interface

Error handling is **not** done via exceptions in Go. Instead, errors are normal *values* of types that implement the built-in `error` interface. The `error` interface is very minimal. It contains only one method `Error()` that returns the error message as a string.

```
type error interface {
  Error() string
}
```

Every time you define a function in which an error could happen during the execution that needs to reach the caller, you need to include `error` as one of the return types. If the function has multiple return values, by convention `error` is always the last one.

```
func DoSomething() (int, error) {
  // ...
}
```

Creating and returning a simple error

You do not have to always implement the error interface yourself. To create a simple error, you can use the `errors.New()` function that is part of the standard library package `errors`. The only thing you need to pass in is the error message as a string, and `errors.New()` will take care of creating a value that contains your message and implements the `error` interface.

If the function returns an error, it is good practice to return the zero value for all other return parameters:

```
func DoSomething() (SomeStruct, int, error) {
  // ...
  return SomeStruct{}, 0, errors.New("failed to calculate result")
}
```

```
You should not assume that all functions return zero values for other return values if ar
It is best practice to assume that it is not safe to use any of the other return values i
The only exceptions are cases where the documentation clearly states that other returns \
```

If you want to use such a simple error in multiple places, you should declare a variable for the error instead of using `errors.New` in-line. By convention, the name of the variable should start with `Err` or `err` (depending on whether it is exported or not). These error variables are often called *sentinel errors*.

```
import "errors"

var ErrNotFound = errors.New("resource was not found")

func DoSomething() error {
  // ...
  return ErrNotFound
}
```

Return `nil` for the error to signal that there were no errors during the function execution:

```
func Foo() (int, error) {
  return 10, nil
}
```

Error checking

If you call a function that returns an error, it is common to store the error value in a variable called `err`. Before you use the actual result of the function, you need to check that there was no error.

To avoid nesting the "happy path" of your code, error cases should be handled first. We can use `==` and `!=` to compare the error against `nil` and we know there was an error when `err` is not `nil`.

```
func processUserFile() error {
        file, err := os.Open("./users.csv")
        if err != nil {
                return err
        }

        // do something with file
}
```

Most of the time, the error will be returned up the function stack as shown in the example above. Another way of handling the error could be to log it and continue with some other operation. It is good practice to either return or log the error, never both.

Since most functions in Go include an error as one of the return values, you will see/use the `if err != nil` pattern all over the place in Go code.

Custom error types

If you want your error to include more information than just the error message string, you can create a custom error type. As mentioned before, everything that implements the `error` interface (i.e. has an `Error() string` method) can serve as an error in Go.

Usually, a struct is used to create a custom error type. By convention, custom error type names should end with `Error`. Also, it is best to set up the `Error() string` method with a pointer receiver, see this Stackoverflow comment to learn about the reasoning. Note that this means you need to return a pointer to your custom error otherwise it will not count as `error` because the non-pointer value does not provide the `Error() string` method.

```
type MyCustomError struct {
  message string
  details string
}

func (e *MyCustomError) Error() string {
  return fmt.Sprintf("%s, details: %s", e.message, e.details)
}
```

```
func someFunction() error {
  // ...
  return &MyCustomError{
    message: "...",
    details: "...",
  }
}
```

Instructions

The day you waited so long finally came and you are now the proud owner of a beautiful farm in the Alps.

You still do not like waking up too early in the morning to feed your cows. Because you are an excellent engineer, you build a food dispenser, the `FEED-M-ALL` .

The last thing required in order to finish your project, is a piece of code that calculates the amount of fodder that each cow should get. It is important that each cow receives the same amount, you need to avoid conflicts. Cows are very sensitive.

Luckily, you don't need to work out all the formulas for calculating fodder amounts yourself. You use some mysterious external library that you found on the internet. It is supposed to result in the happiest cows. The library exposes a type that fulfils the following interface. You will rely on this in the code you write yourself.

```
type FodderCalculator interface {
        FodderAmount(int) (float64, error)
        FatteningFactor() (float64, error)
}
```

As you work on your code, you will improve the error handling to make it more robust and easier to debug later on when you use it in your daily farm live.

1. Divide the food evenly

First of all, you focus on writing the code that is needed to calculate the amount of fodder per cow.

Implement a function `DivideFood` that accepts a `FodderCalculator` and a number of cows as an integer as arguments. *For this task, you assume the number of cows passed in is always greater than zero.* The function should return the amount of food per cow as a `float64` or an error if one occurred.

To make the calculation, you first need to retrieve the total amount of fodder for all the cows. This is done by calling the `FodderAmount` method and passing the number of cows. Additionally, you need a factor that this amount needs to be multiplied with. You get this factor via calling the `FatteningFactor` method. With these two values and the number of cows, you can now calculate the amount of food per cow (as a `float64`). That is what should be returned from the `DivideFood` function.

If one of the methods you call returns an error, the execution should stop and that error should be returned (as is) from the `DivideFood` function.

```
// For this example, we assume FodderAmount returns 50
// and FatteningFactor returns 1.5.
DivideFood(fodderCalculator, 5)
// => 15 <nil>

// Now assuming FodderAmount returns an error with message "something went wrong".
DivideFood(fodderCalculator, 5)
// => 0 "something went wrong"
```

2. Check the number of cows

While working on the first task above, you realized that the external library you use is not as high-quality as you thought it would be. For example, it cannot properly handle invalid inputs. You want to work around this limitation by adding a check for the input value in your own code.

Write a function `ValidateInputAndDivideFood` that has the same signature as `DivideFood` above.

- If the number of cows passed in is greater than 0, the function should call `DivideFood` and return the results of that call.
- If the number of cows is 0 or less, the function should return an error with message `"invalid number of cows"`.

```
ValidateInputAndDivideFood(fodderCalculator, 5)
// => 15 <nil>
```

```
ValidateInputAndDivideFood(fodderCalculator, -2)
// => 0 "invalid number of cows"
```

3. Improve the error handling

Checking the number of cows before passing it along was a good move but you are not quite happy with the unspecific error message. You decide to do better by creating a custom error type called `InvalidCowsError`.

The custom error should hold the number of cows (`int`) and a custom message (`string`) and the `Error` method should serialize the data in the following format:

```
{number of cows} cows are invalid: {custom message}
```

Equipped with your custom error, implement a function `ValidateNumberOfCows` that accepts the number of cows as an integer and returns an error (or nil).

- If the number of cows is less than 0, the function returns an `InvalidCowsError` with the custom message set to `"there are no negative cows"`.
- If the number of cows is 0, the function returns an `InvalidCowsError` with the custom message set to `"no cows don't need food"`.
- Otherwise, the function returns `nil` to indicate that the validation was successful.

```
err := ValidateNumberOfCows(-5)
err.Error()
// => "-5 cows are invalid: there are no negative cows"
```

After the hard work of setting up this validation function, you notice it is already evening and you leave your desk to enjoy the sunset over the mountains. You leave the task of actually adding the new validation function to your code for another day.

Source

Created by

- @brugnara

- @jmrunkle
- @junedev